Traceability for Mutation Analysis in Model Transformation

Vincent Aranega, Jean-Marie Mottu, Anne Etien, Jean-Luc Dekeyser

Traceability for Mutation Analysis in Model Transformation

Vincent Aranega[1], Jean-Marie Mottu[2], Anne Etien[1], and Jean-Luc Dekeyser[1]

[1] LIFL - UMR CNRS 8022, INRIA, University of Lille 1
Lille, France
firstname.lastname@lifl.fr
[2] LINA - UMR CNRS 6241, University of Nantes
Nantes, France
jean-marie.mottu@univ-nantes.fr

Abstract. Model transformation can't be directly tested using program techniques. Those have to be adapted to model characteristics. In this paper we focus on one test technique: mutation analysis. This technique aims to qualify a test data set by analyzing the execution results of intentionally faulty program versions. If the degree of qualification is not satisfactory, the test data set has to be improved. In the context of model, this step is currently relatively fastidious and manually performed.
We propose an approach based on traceability mechanisms in order to ease the test model set improvement in the mutation analysis process. We illustrate with a benchmark the quick automatic identification of the input model to change. A new model is then created in order to raise the quality of the test data set.

1 Introduction

When a program written in C has not the expected behavior or is erroneous, the programmers look for the faults in their program. Indeed, they trust in the compiler. The C compilers have been largely tested for two major reasons. First, a fault in a compiler may spread over lot of programs since a compiler is used many times to justify the efforts relative to its development. Secondly, compilers have to be trustworthy. Indeed, when the execution of a C program leads to an unexpected behavior, the faults have to be looked for in the program and not in the compiler. Similarly, model transformations that form the skeleton of model based system development, and so enable to generate code from high level model specifications, have to be largely tested and trustworthy.

Model transformations may be considered programs and tested as such. However, the data structures they manipulate (models conform to metamodels) implies specific operations that do not occur in traditional programs such as navigating the metamodels or filtering model elements in collections. Thus, classical but also specific faults may appear in model transformations. For instance, the programmer may have navigated a wrong association between two classes, thus manipulating incorrect class instances of the expected type. The emergence of

the object paradigm has implied an evolution in the verification techniques [13]. Similarly, verification techniques have to be adapted to model transformation specificity to make profit from the model paradigm. New issues relative to the generation, the selection and the qualification of input model data are met.

There exist several test techniques. In this paper, we only focus on mutation analysis. This technique relies on the following assumption: if a given test data set can reveal the fault in voluntarily faulty programs, then this set is able to detect involuntary faults. Mutation analysis [6] aims to qualify a test data set for detecting faults in a program under test. For this purpose, faulty versions of this program (called *mutants*) are systematically created by injecting one single fault by version. The efficiency of a given test data set to reveal the faults in these faulty programs is then evaluated. If the proportion of detected faulty programs [20] is considered too low, new data tests have to be introduced [16].

Only the test data improvement step of the mutation analysis process dedicated to model transformation is apprehended in this paper. Indeed, in [12], the authors argue, with a survey of the development of mutation testing, that few works deals with that test set improvement step. The creation of new test models relies on a deep analysis of the existing test models and the execution of the unrevealed faulty transformations. Currently, this work is manually performed and fastidious; the tester deals with a large amount of information. Thus, in this paper, we propose an approach to fully automate the information collection. This automation relies on traceability mechanisms enhanced with mutation analysis characteristics. An algorithm is proposed to effectively collect the required and sufficient information. Then, the collected information is used to create new test models. Our enhanced traceability mechanisms helps to reduce the testers intervention to particular steps where their expertises are essential.

This paper is composed as follows. Section 2 presents mutation analysis to qualify test data set in model transformation testing. Section 3 describes our metamodels, foundations of our approach to improve test data set. Section 4 validates our approach with the *class2table* transformation. Section 5 introduces works related to the qualification and the improvement of the test data set. Section 6 draws some conclusions and introduces future work.

2 Mutation Analysis to Qualify Test Data Set

Assuring that a program is undoubtedly fault free is a difficult task requiring a lot of time and expertise. However, qualifying a test data set (*i.e.* estimate its pertinence and its effectiveness) is easier. If this estimation is considered too low, the test set must be improved. In the following subsections, we briefly describe the mutation analysis process [6], one way to qualify a given test data set. We then explain why that software testing method has to be adapted to the model paradigm.

2.1 Mutation Analysis Process

The mutation analysis process may be divided into four activities as sketched in Figure 1. The preliminary step (*i.e.* activity (a)) corresponds to the definition of an initial test set, that the tester wants to qualified and the creation of variants (P_1, P_2,..., P_k) (called mutants) of the program P under test by injecting one atomic change. In practice, each change corresponds to the application of a single mutation operator on P. Then, P and all the mutants are successively executed with each test data of the set that has to be qualified (*i.e.* activity (b)).

If the behavior of P with one of the test data differs from anyway from the behavior of at least one of the P_i with the same test data, these mutants are said to be *killed*. The faults introduced in those P_i were indeed highlighted by the test data. In the other case, if P returns the same results as some P_j, they are said to be *live* mutants. The activity (c) computes the ratio of killed mutants also called the mutation score. If this ratio is considered too low, this means that the test data set is not sensitive enough to highlight the faults injected in the program. In that latter case, the test data set has to be improved (activity (d)) until it kills each mutant or it only leaves live mutants that are equivalent to P [6]; *i.e.* no test data can distinguish P and these live mutants (*e.g.* the fault is inserted in dead code).

The mutation analysis process is stopped when the test data set is qualified *i.e.* when the mutation score reached 100 % or when it rose above a threshold beforehand fixed.

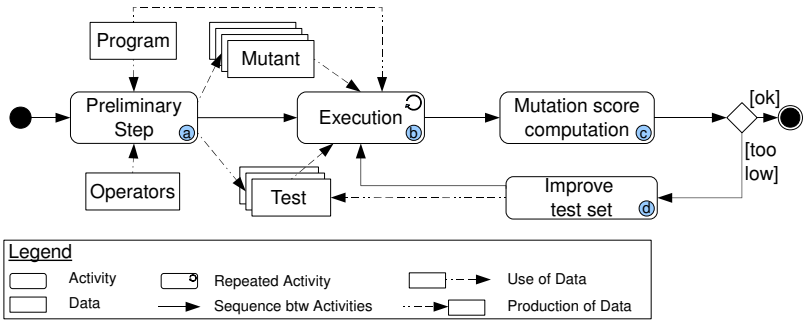

Fig. 1. Mutation analysis process

2.2 A largely manual process

Part of the mutation analysis process is automatic but work remains for the tester. The mutant creation can be automated. However, usually the operators are specific to the language used in the program to test. For each new language the mutation operators have to be defined and implemented. The execution of P and its associated mutants with the test data is obviously automated as well as the comparison of the outputs. The analysis of a live mutant is manual up to now.

Indeed, on the one hand, the automatic identification of equivalent mutants is an undecidable problem [6, 17]. On the other hand, the test data set improvement can be difficult. The improvement of the test data set is manually performed. Indeed, the unrevealed injected fault should be analyzed both statically and dynamically in order to create a new test data that will kill the considered mutant.

The purpose of this paper is to help in the automation of the test data set improvement in case where test data are models and program is a model transformation. But let us explore in the next subsection the specificity of model transformation testing.

2.3 Adaptation to Model Transformation

Model transformations can be considered programs and therefore techniques previously explained can be used. However, the complexity and the specificity induced by the data structures (*i.e.* models conform to their metamodels) manipulated by the transformations imply modifications in the mutation analysis process described in the subsection 2.1.

Each step of the mutation analysis process has to be adapted to model transformations. [18] deals with the generation of test models. In [14], dedicated mutation operators have been designed independently from any transformation language. They are based on three abstract operations linked to the basic treatments of a model transformation: the navigation of the models through the relations between the classes, the filtering of object collections, and the creation and the modification of the model elements. The execution of the transformation under test T and its mutants $T_1, T_2, \ldots, T_k$ differs from the execution of a program but remains common. The comparison of the output model produced by T and those produced by the T_i can be performed using adequate tools such as EMFCompare [1]. If a difference is raised by EMFCompare, the mutant is considered *killed*, otherwise new test models are built to kill the (non equivalent) live mutants.

The remainder of this paper focuses on the improvement of the test set (activity (d)) in the mutation analysis process dedicated to model transformation. Our proposition relies on the following hypothesis: Building new test models from scratch can be complex whereas creating a new test model could benefit from the existing models. Thus we have developed an approach that creates new test model by adaptation of other existing and pertinent ones.

3 Traceability, a Means to Automatically Collect Information

Considering that creating a new test model from another one is easier than from scratch, the issue of the test set improvement raises three questions:

- Among all the existing couples (test model, mutant), which ones are relevant to be studied?

– What should the output model look like if the mutant was killed? *i.e.* what could be the difference we want to make appear in the output model?
– How to modify the (input) test model to produce the expected output model and thus kill the mutant?

To help the tester to answer these questions, we provide a method based on a traceability mechanism.

3.1 Traceability for Model Transformation

According to the IEEE Glossary, *Traceability allows one to establish degrees of relationship between products of a development process, especially products bound by a predecessor-successor or master-subordinate relationship* [11]. Regarding MDE and more specifically model transformations, the trace links elements of different models by specifying which ones are useful to generate others.

Our traceability approach [9, 2] relies, among others, on the local trace metamodel presented in Figure 2.

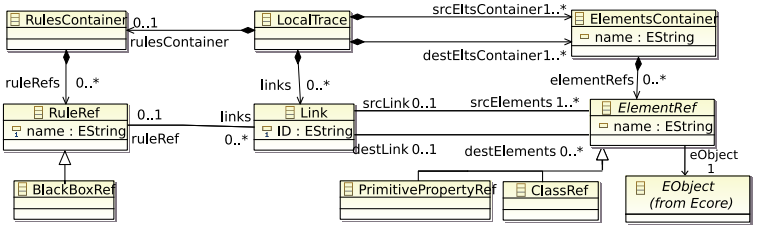

Fig. 2. Local Trace Metamodel

The local trace metamodel is built around two main concepts: *Link* and *ElementRef* expressing that one or more source elements are possibly bound to target elements. Furthermore, for each link, the transformation rule producing it is traced using the *RuleRef* concept. Finally, for implementation facilities, an *ElementRef* has a reference to the real object in the source or target model. As our environment is based on the Eclipse platform, models are implemented with EMF, the reference named *EObject* is an import of the *ECore* metamodel. The local trace metamodel and local trace models are independent of any transformation language. However, the generation of the local trace model strongly depends on the used transformation language.

For each *Link* instance, the involved elements of the input or output models are clearly identified thanks to the *ClassRef* directly referring the *EObject*. A continuity between the traceability and the transformation worlds is ensured. Furthermore, the transformation rule that has created a link is associated to it via the *ruleRef* reference. Each time a rule is called a unique new *Link* is created. Thus, from a rule, the *localTraceModel* enables the tester to identify, for each call (*i.e.* for each associated link), two sets of elements: those of the input model

and those of the output model created by the rule. In the case of a faulty rule, these sets respectively correspond to the elements to modify and the elements that may be different if the mutant is killed.

3.2 Mutation Matrix Metamodel

Mutation analysis results and traces information are combined to automate a part of the test data set improvement process. Mutation analysis results are usually gathered in a matrix. Each cell indicates if an input model has killed a given mutant or not. A mutant is alive if none of its corresponding cells indicate a killing. From information contained in all the cells concerning a mutant, it can be deduced if it is alive or not.

Links between mutants, test models and their traces are managed using a dedicated matrix at a model level. The advantages are multiple. A cell corresponds to an abstraction of the execution of a mutant T_i for the test model D_k. By associating its trace to each cell, the matrix model becomes a pivot model. In this way a continuity is ensured between the traces, the test models and the information gathered in the mutation matrix. The navigation is eased between the different worlds. Moreover, the mutation matrix benefits from tools dedicated to models. Thus, the mutation matrix model is automatically produced from the results of the comparisons between the model produced by the original transformation and the one generated by T_i.

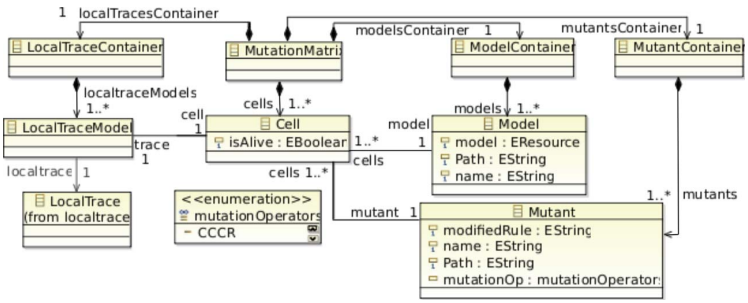

Fig. 3. Mutation Matrix Metamodel

The mutation matrix metamodel, presented in Figure 3, is organized around three main concepts. *Mutant* refers to mutants created from the original transformation. The mutants have one rule ($modifiedRule$ attribute) modified thanks to one mutation operator ($mutationOp$ attribute). *Model* refers to input test data. *Cell* corresponds to an abstraction of the couple (mutant T_i, test model D_k). Its value ($false$ or $true$) of the property $isAlive$ specifies the state (killed or live respectively) of the *Mutant* T_i regarding to the specific *Model* D_k. The *LocalTraceModel* corresponding to the execution of T_i with D_k is thus associated to the *Cell*.

The matrix model is generated during the mutation analysis process. It is the doundation of the test model improvement process presented in the next subsection.

3.3 Data Improvement Process Assisted by Traces

This section aims to clarify and expose the data improvement process enhanced with our traceability mechanism. An overview of our proposition is shown before detailing the different steps of this process.

Overview The data improvement process (activity (d) in Figure 1) is composed of three activities as shown in Figure 4: (1) the selection of a live mutant, (2) the identification of a relevant test model and (3) the creation of a new test model by adaptation of the existing test model previously identified. These three activities rely on either the mutation matrix, the trace model or both. Indeed, in the mutation matrix, each cell corresponds to a couple (mutant, test model). The results of the execution of the considered mutant with the model in question are gathered in a trace model. We developed some algorithms to scan the mutation matrix and the trace model in order to gather adequate information. In this paper we focus on the second activity and give some exploratory ideas on the third one.

Step 1: Selection of a Live Mutant A mutant is alive if no test model has killed it. Live mutants can thus be easily and automatically identified by exploring the matrix cells. Each cell relative to a mutant is scanned. If for all of them the property *isAlive* is set to *true*, the mutant is considered alive.

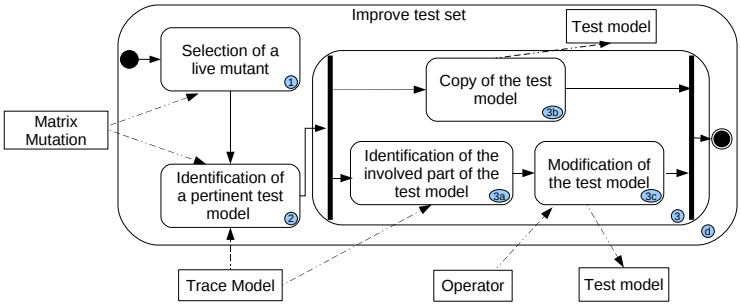

Fig. 4. Test Model Improvement Process

Step 2: Identification of a Relevant test model Identifying a good candidate, among the test models, to kill a given live mutant is more difficult. Our approach relies on the principle that test models for which the faulty rule of the mutant has been called are better candidates. Indeed, the conditions to apply

this rule were satisfied. Our traceability mechanism helps us to identify these models and for each of them to highlight the elements impacted by the faulty rule. The algorithm 1 implements this part of the improvement process (*i.e.* corresponding to step 2 and gathering information to perform step 3).

Algorithm 1 Information Recovering for a Live Mutant

```
1: trace ← null
2: rule ← null
3: modifiedRule ← mutant.modifiedRule
4: modelsHandled ← ∅
5: eltsHandledSrc ← ∅
6: eltsHandledDest ← ∅
7: for each mutant.cells do
8:     trace ← cell.trace
9:     rule ← trace.findRule(modifiedRule)
10:    if rule ≠ null then
11:        modelsHandled += cell.model
12:        tempEltsSrc ← ∅
13:        tempEltsDest ← ∅
14:        for each rule.links do
15:            tempEltsSrc += link.srcElements
16:            tempEltsDest += link.destElements
17:        end for
18:        eltsHandledSrc += tempEltsSrc
19:        eltsHandledDest += tempEltsDest
20:    end if
21: end for
```

The first five lines correspond to the initialization of the different variables. The *trace* variable stores the trace associated to the execution of the mutant T_i for a given test model D_k. *rule* refers to a RuleRef in the trace model. $modifiedRule$ is a String initialized with the name of the modified rule associated to T_i. The $modelsHandled$ variable is a model list containing the test models for which the execution of the mutant requires the modified rule. The $eltsHandledSrc$ and $eltsHandledDest$ variables are similar to the previous one. They contain lists of input (respectively output) elements (one list by test model) that are involved in the application of the faulty rule.

The algorithm then scans each cell relative to the studied mutant. The trace corresponding to the execution of T_i on one input model D_k is stored (line 8). The trace model is navigated to check if the modified rule has been called during the corresponding transformation. This search is performed through the $findRule$ method (not detailed in the algorithm). This method explores the *RulesContainer* of the *LocalTraceModel* associated to the cell until it finds the *RuleRef* instance whose name corresponds to the one of the faulty rule (*i.e.* the assigned value of the modifiedRule property of the *Mutant*). This method returns

a *RuleRef* instance or null if the rule doesn't appear in the trace. The result is stored in the *rule* variable (line 9). If the content of the *rule* variable is *null*, the analysis stops here for this cell and goes on with the next one. On the other hand, the model D_k is stored in the $modelHandled$ (line 10). For each link associated to the *rule*, the list of the input model elements (*srcElements*) is stored in the $eltsHandledSrc$ variable using the temporary variable *tempEltsSrc*. The management of the output model elements is performed from the same way. (line 12 to 17).

For a given live mutant T_i, this algorithm provides: (1) some test models ($modelsHandled$) (2) their elements ($eltsHandled$) involved in the application of the faulty rule and (3) the elements of the output models created by this rule. If the content of the $modelHandled$ variable is empty, the faulty rule has never been called, whatever the test model. A new model has to be created, possibly from scratch, containing elements satisfying the application of the faulty rule. On the other hand, if the $modelHandled$ variable is not empty, the faulty rule has been called at least once. However, since the mutant is alive, this rule has never produced a result different from the one generated by the original transformation T. A new test model is created by adapting the considered test model.

Step 3: Creation of a New Test Model During this step, a new test model is produced using the information gathered in the previous one. The step is composed of three sub-steps. The most important ones are the $3(a)$ and $3(c)$ (figure 4), that modify a test model previously identified as relevant. As these sub-steps deliberately modify a chosen test model among the existing ones, the sub-step $3(b)$ copies the considered test model in order to conserve it unchanged in the produced new test model set.

The modification of the test model requires both the previously identified elements and the applied operator. Indeed, based on a static analysis, the tester must understand why the mutant remains alive whereas the mutated rule has been called on the identified elements. He then must consequently modify the model. These two activities are currently manually performed. However, we observe that for each mutation operator, the number of situations letting the mutant alive is low. We have initiated to list, for each operator, all these situations and identify some related modifications to perform on the model to kill the mutant. For each mutant, the list of the situations must be exhaustive, but for each of these situations, only one modification enabling the tester to kill the mutant is enough. We foresee to develop a tool that, given an operator and the identified relevant model, will automatically detect one of these situations and perform the modification on this model.

The *RSMA* mutation operator [14] is taken for the following example. It adds a useless navigation to an existing navigation sequence while respecting the metamodel involved in the transformation. Thus, for example, the original transformation navigates the sequence $self.a.b$ and the mutant navigates $self.a.b.c$. In only three cases, the mutant may remain alive: (1) the original and the mutated navigation sequence finally point to the same instance; (2) the original and

the mutated navigation sequence finally point to *null*; (3) the property values pointed by the original and the mutated navigation sequence are the same. The way to modify the test model differs in each of these three cases. The first case occurs when the added navigation is the same that the last one in the original sequence. Such a situation is possible if the added navigation corresponds to a reflexive reference in the metamodel. The mutated navigation is thus $self.a.b.b$ (for the original navigation sequence $self.a.b$). The mutant can be killed if the original and the mutated navigation point to two different instances of the same metaclass. A new instance (with different properties) of this metaclass must thus be added in the test model and references consequently updated. The second case occurs if one intermediate navigation is not set. In our example, if the a reference points to *null*, neither the original navigation sequence $self.a.b$ nor the mutated one $self.a.b.c$ can be fully performed, the object retrieved in both sequences is *null*. The test model can be modified in order to fill in the empty references. The third case occurs when the class recovered by the sequence $self.a.b$ and the class recovered by the sequence $self.a.b.c$ both own a property with the same label (*e.g.* the property *name*). The mutant remains alive if, in the test model, these properties are set to the same value for the two recovered instances (of possibly two different metaclasses). To solve this problem, the testers can modify one of the properties changing its value.

By extending such a work on all the operators identified in [14], the creation of a new test model can be even more automated. However, the algorithm underlying this automation will rely on the used transformation language. In order to capitalize this work whatever the used transformation language it seems inevitable to use generic definition of mutation operators. In [19] the authors propose MuDeL, a language enabling the description of mutant operators independently from the used language. Thus a given generic operator can be reused with several languages. However, the MuDeL operators are dedicated to traditional programs and not to model transformations. A generic representation of the mutation operators defined in [14], would largely benefit to the independence of our approach to any transformation language.

4 Example

This section aims to validate our approach on a case study; the classical UML to Relation Data Base Management Systems (RDBMS) transformation (*class2-rdbms*). For the example, we used a simple version of the UML class diagram (simpleCD) and a simple version of the *class2rdbms* transformation. The transformation specification we adopt is the one proposed at the MTIP workshop [5]. We have implemented this transformation with Kermeta [15]. The transformation counts around 113 lines of code in 11 operations. The choice of the Kermeta language results in the work initiated in [14]. Using the same transformation and mutant enables us to compare and evaluate the approach proposed in this paper.

4.1 Application of our Approach

For the experimentation, 200 mutants have been manually created (105 for the *navigation* category, 75 for the *filtering* category and 20 for the *creation* category). Initially, 16 test models have been defined. The mutants are executed. The mutation matrix is filled based on the model comparison. Then, the mutation matrix is automatically explored in order to identify the alive and the killed mutants. The remainder of the algorithm is applied for an alive mutant. The first alive mutant identified is the *navigation/Class2RDBMS_19.kmt*. Listing 1.1 represents an excerpt of this mutant. The original piece of code is marked by the *orig* flag and the modified one by the *mutant* flag. Initially, the transformation fills in the *prefix* attribute of the *FKey* class by concatenating a variable with the *name* attribute of the *Association*. The mutant concatenates the same variable with the *name* of the *dest* attribute belonging to the *Association*. A navigation has been introduced in an existing sequence using the *RSMA* mutation operator (Relation Sequence Modification with Addition) [14].

```
1  operation createColumnsForAssociation( ...,
2  asso : Association, prefix : String) is
3  do
4  ...
5  var fk : FKey init FKey.new
6  //fk.prefix := prefix + asso.name // orig
7  fk.prefix := prefix + asso.dest.name // mutant
8  ...
9  end
```

Listing 1.1. Mutate *createColumnsForAssociation* rule excerpt

Thanks to the mutation matrix, the mutated rule is recovered from the *modifiedRule* property of the *Mutant* Class. For the studied mutant, the rule *createColumnsForAssociation* is immediately identified.

The algorithm 1 identified 7 test models that have triggered the mutated rule (the *createColumnsForAssociation* rule). The algorithm also provides the elements of the identified models (the $eltsHandledSrc$ set) involved by the application of this rule. Moreover, the elements created in the output model from the elements that reach the mutated rule are, also, highlighted and gathered in the $eltsHandledDest$ set. Table 1 gathers the results. For example, the *classModel04.simpleuml* model triggers the mutated rule that only handles the *Customer:Association* element and produces the *Customer:FKey* element in the destination model.

A quick static analysis of the mutated rule indicates that the *prefix* is formed using the *name* of the *dest* instead of the *name* of the *Association* directly. Based on these information and on the algorithm results, the remainder of the improvement test set process is manually performed. The *prefix* property of the *FKey* is set to *Customer*. The same occurs for the model produced by the original transformation. Thus, in order to kill the mutant, the model created by the mutated transformation must provide a different value for the *prefix* property. As no difference are raised, the tester can infer that in the 7 identified models, the *name* of the *Association* is the same as the *name* of the element pointed by *dest*. Easily, a new test model is created by modification of an existing one (for

Table 1. Test Model Elements Handled by the Modified Rule

Test Model	**Src. Elements** ($eltsHandledSrc$)	**Dest. Elements** ($eltsHandledSrc$)
ClassModel02.simpleuml	*c:Association*	*c:FKey*
ClassModel03.simpleuml	*b:Association*	*b:FKey*
ClassModel04.simpleuml	*Customer:Association*	*Customer:FKey*
ClassModel05.simpleuml	*blah:Association*	*blah:FKey*
ClassModel06.simpleuml	*c:Association*	*c:FKey*
ClassModel07.simpleuml	*c:Association* *b:Association*	*c:FKey* *b:FKey*
ClassModel08.simpleuml	*a:Association* *b:Association*	*a:FKey* *b:FKey*

example: *ClassModel04.simpleuml*). This model is copied, then the *name* of the *Association* is changed from *Customer* to *CustomerAssoc*.

In order to check the efficiency of the new test, the mutation analysis process is performed once again. This time, 17 models are taken in account. The studied mutant is henceforth killed and this same added model also killed 2 other mutants that probably modified a rule using in the same way as the studied mutant. Once the mutation analysis is played again with the new test model set, the process goes on with another live mutant.

The modifications to perform on the test model to create a new one are not so easy than the one of the above example. However, this example illustrates the relevance and the usefulness of the information gathered thanks to our algorithm in order to raise the quality of a test model set.

4.2 Quantitative Study

This section aims to show that our approach enables the tester to save a considerable amount of time and that the execution time remains largely acceptable whereas 3200 executions are performed and so many results analyzed. For this purpose, we perform different benchmarks corresponding to 8, 9 and 16 test models, respectively.

Identification of the live mutants. The number of mutants remains 200 in the three benchmarks. The only variable parameter is the number of input model and thus the number of cells to explore for each mutant. The live mutant identification only uses the mutation matrix that is loaded once. The loading time is closely bound to the mutation matrix size. The loading time was short and approximated 1 second. The mutation matrix contains around 3200 cells + 3200 traces, corresponding to a loading time of $5107ms$. Then, once the mutation matrix is loaded, the operations performed in order to identify the live mutants are only navigations. Fortunately, this kind of operations are quite instantaneous, and the observed execution time are lower than 1 second for each benchmark (for the bag of 16 test models, the algorithm identified 24 live mutants in $461ms$).

Execution of the algorithm 1 for one live mutant. This part aims to measure the execution time of the algorithm that identifies the useful tests mod-

els. Three benchmarks with respectively 8, 9 and 16 test models are performed[3]. The algorithm identifies three elements: potential useful models, input elements and output elements impacted by the application of the faulty rule. In the three cases, the algorithm identifies the 7 models in $308ms$ for the set of 8 and 9 test models and $311ms$ for the set of 16 test models (Table 2). The fact that the execution time is slightly different in the latter case corresponds to the scan of the traces relative to the seven more test models (the mutation matrix is considered already loaded). Of course, these measures depend on the test model and the trace sizes, but they are largely inferior to the time spend to manually collect the information. Going in details with these results also shows that the 7 more test data added to the set are not useful for this mutant and require a more complex modification in order to reach the mutated rule and kill the mutant. Nevertheless, their presence in the test model set is relevant because they allow to kill other mutants.

Table 2. Identified models for each test models set

Test set size	**Number of identified models**	**Execution time**
8	7	$308ms$
9	7	$308ms$
16	7	$311ms$

Our approach has to be tested with hundreds test models and the execution time measured. However, the quantitative analysis is promising concerning the scalability of our algorithm.

Comparison with other models Our approach has also been used with model transformations written in QVTO [3], in the context of the Gaspard 2 framework. This framework aims to generate, from a UML model enhanced with the profile dedicated to real time and embedded systems, programs in various languages depending on the purpose (simulation, execution, verification ...). The order of magnitude were approximately the same. The loading time of the matrix was smaller (around 1 second) because of the matrix size (only 1120 cells + 1120 traces). However, the execution times of the algorithm were higher because the models contained much more number of classes and referenced a UML profile.

Comparison with the manual process The test set improvement is a hard and complex task for the testers. They have to perform static analysis to identify why a mutant has not been killed. However, they may do this analysis with test model, without leading to any relevant results. Indeed, some existing models have to be heavily modified before killing a mutant. Manually identifying an adequate test model from which it will be easy to create a new one killing the mutant may be very long. For the *class2table* transformation, the manual information collection can take from few minutes for some easy cases to few hours for the most complex ones. Using our algorithm allows the testers to recover the same piece of information in less than a minute.

[3] On a DELL Precision 490/Gentoo-2.6.34

5 Related Work

There are different ways to obtain a qualified test data set. Since model transformation testing has only been briefly studied, few works consider test models qualification and improvement.

Fleurey et al. [7] propose to qualify a set of test models regarding its coverage of the input domain. The input domain is defined with metamodels and constraints. The qualification is static and only based on the input domain whereas the mutation analysis relies on a dynamic analysis of the transformation. In case of very localized transformations, the approach developed by Fleurey et al. produces more models than necessary.

However, in [8], they also propose an adaptation of bacteriologic algorithm to model transformation testing. The bacteriologic algorithm [4] is designed to automatically improve the quality of a test data set. It measures the mutation score of each data to (1) reject useless test data, (2) keep the best test data, (3) "combine" the latter to create new test data. Their adaptation consists in creating new test models by covering part of the input domain still not covered. The authors use the bacteriologic algorithm to select models whereas we propose the mutation analysis associated to trace mechanisms.

In [10], authors study how to use traceability in test driven development (TDD). TDD involves writing the tests prior to the development of the system. Here, traceability can be used to help the creation of new tests considering how the system covers the requirements. The trace links the requirement and the code, and helps the developer to choose the next features which should be tested, then coded. In that approach they do not consider the fault revealing power of the test data set, but the coverage of the requirements to assist the creation of test data.

6 Conclusion

As any other program, it is important to test model transformations. For this purpose, test data set has to be qualified. Mutation analysis is an existing approach that has already been approved and adapted to model transformations. In this paper, we focus on the test model set improvement step and propose a traceability mechanism in order to ease the tester job. This mechanism completely adopts the model paradigm and relies on a local trace metamodel and a matrix metamodel.

Our approach helps the tester to drastically reduces the field of the required analysis to create a new model. We have shown on the *RSMA* operator that the number of situations where a mutated rule is executed for a test model while letting the mutant alive is low. The modifications to performed in those cases are well identified. We are currently working on a generic representation of the mutation operators in order to go towards one step further in the automation the mutation analysis process and to remain independent from the used transformation language.

References

1. EMFcompare. www.eclipse.org/emft/projects/compare.
2. V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Traceability mechanism for error localization in model transformation. In *ICSOFT*, Bulgaria, July 2009.
3. V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Using traceability to enhance mutation analysis dedicated to model transformation. In *Workshop MoDeVVa 2010 associated with Models2010 conference*, Oslo, Norway, Oct. 2010.
4. B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *STVR Journal*, 15(2):73–96, 2005.
5. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In *Satellite Events at the MoDELS 2005 Conference*, 2005.
6. R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.
7. F. Fleurey, B. Baudry, P.-A. Muller, and Y. Le Traon. Towards dependable model transformations: Qualifying input test data. *SoSyM Journal*, 2007.
8. F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *Proceedings of MoDeVVa*, pages 29–40, Nov. 2004.
9. F. Glitia, A. Etien, and C. Dumoulin. Traceability for an MDE Approach of Embedded System Conception. In *ECMDA Traceability Workshop*, Germany, 2008.
10. J. H. Hayes, A. Dekhtyar, and D. S. Janzen. Towards traceable test-driven development. In *TEFSE Workshop*, pages 26–30, USA, 2009. IEEE Computer Society.
11. IEEE. *IEEE standard computer dictionary : a compilation of IEEE standard computer glossaries*. IEEE Computer Society Press, New York, NY, USA, 1991.
12. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, To appear, 2010.
13. Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.
14. J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *ECMDA 06*, Spain, July 2006.
15. P. Muller, F. Fleurey, and J. Jzquel. Weaving executability into objectoriented meta-languages. In S. K. L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume 3713 of *LNCS*, pages 264–278, Montego Bay, Jamaica, Oct. 2005.
16. T. Murmane, K. Reed, T. Assoc, and V. Carlton. On the effectiveness of mutation analysis as a black box testing technique. In *Software Engineering Conference*, pages 12–20, 2001.
17. A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.
18. S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *ICST.*, Norway, Apr. 2008.
19. A. Sim ao, J. C. Maldonado, and R. da Silva Bigonha. A transformational language for mutant description. *Comput. Lang. Syst. Struct.*, 35(3):322–339, 2009.
20. J. M. Voas and K. W. Miller. The revealing power of a test case. *Softw. Test., Verif. Reliab.*, 2(1):25–42, 1992.